NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Approved for public release; distribution is unlimited.

20010905 133

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 2001	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Supporting The Secure Halting Of User Sessions And Processes In The Linux Operating System	5. FUNDING NUMBERS
6. AUTHOR(S) Brock, Jerome P.	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

One feature of a multi-level operating system is a requirement to manage multiple, simultaneous user-sessions at different levels of security. This session management is performed through a trusted path between the user and operating system. Critical to this functionality is the operating system's ability to temporarily halt dormant sessions, thereby ensuring their inability to perform any actions within the system. Only when a session must be reactivated are its processes returned to a runnable state.

This thesis presents an approach for adding this "secure halting" functionality to the Linux operating system. A detailed design for modifying the Linux kernel, the core of the operating system, is given. A new module, allowing an entire session to be halted and woken up, is designed. A new process state, the "secure halt" state, is added. Additionally, the kernel's scheduling manager is modified to properly manage processes in the secure halt state. The research has led to the implementation of the design as a proof of concept.

This research is meant to be used in combination with other efforts to enhance the security of the Linux operating system.

14. SUBJECT TERMS Secure Halt, Trusted Path, Secure Attention Key, Linux, Computer Security	15. NUMBER OF PAGES 76
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFI- CATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

SUPPORTING THE SECURE HALTING OF USER SESSIONS AND PROCESSES IN THE LINUX OPERATING SYSTEM

Jerome P. Brock
Captain, United States Army
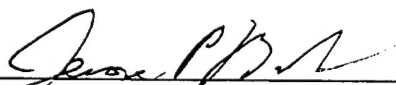B.S., United States Military Academy, 1991

Submitted in partial fulfillment of the
requirements for the degree of
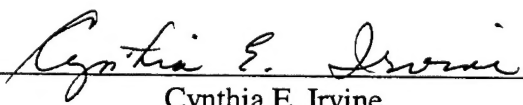
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2001

Author: _____
Jerome P. Brock

Approved by: _____
Paul C. Clark

_____
Cynthia E. Irvine

_____
Dan Boger, Chairman
Department of Computer Science

iv

ABSTRACT

One feature of a multi-level operating system is a requirement to manage multiple, simultaneous user-sessions at different levels of security. This session management is performed through a trusted path between the user and operating system. Critical to this functionality is the operating system's ability to temporarily halt dormant sessions, thereby ensuring their inability to perform any actions within the system. Only when a session must be reactivated are its processes returned to a runnable state.

This thesis presents an approach for adding this "secure halting" functionality to the Linux operating system. A detailed design for modifying the Linux kernel, the core of the operating system, is given. A new module, allowing an entire session to be halted and woken up, is designed. A new process state, the "secure halt" state, is added. Additionally, the kernel's scheduling manager is modified to properly manage processes in the secure halt state. The research has led to the implementation of the design as a proof of concept.

This research is meant to be used in combination with other efforts to enhance the security of the Linux operating system.

TABLE OF CONTENTS

x

I.   INTRODUCTION

A.   COMPUTER SECURITY

Computers are everywhere. They provide useful services in an almost unlimited range of areas. Given their widespread use, the importance of computer security cannot be understated. Initial attempts at computer security revolved around physically protecting the computer. Early computers were rare, large, expensive machines. One user at a time would have access to the computer. There was no need to protect one user's data from another's as only one user's data was on the computer at any one time. Since then, the method of computer use has changed dramatically.

Modern computers are shared by multiple users. This necessitates protecting one user's data from another's. This duty falls on the computer's operating system. An operating system is a computer program that sits between the user's programs/data and the hardware. One of its primary responsibilities is to accept user requests and decide the most efficient manner in which to allocate its resources in order to accomplish the request. As part of this responsibility, the operating system must decide if the user is authorized access to the requested resource. This duty is known as access control.

B.   ACCESS CONTROL

Modern operating systems provide some methodology of protecting users from each other and the operating system from the users. This protection is called access control. In general, access control requires that the operating system mediate all requests by a user wishing to utilize a resource. In a computer, a process executing on behalf of a

user is known as a subject. An object is something on the system that must be protected, i.e. files, printers, ports, etc.

The reference validation mechanism (RVM) is responsible for validating all requests by a subject to access an object. The RVM does this by comparing the security attributes of the subject with those of the object. If the access is acceptable in accordance with the RVM's rule set, the RVM will allow the access. RVM rule sets fall into two major categories: mandatory access control and discretionary access control.

1.   Mandatory Access Control

In a mandatory access control (MAC) system, each subject and object's security attributes are unchangeable. Certain individuals, usually the system administrators, are responsible for specifying the security attributes of a newly created user ID. Those security attributes specify a range of values. When a user logs on, he selects a value from this range. This selected value is given to the user's initial subject when he first logs in.

2.   Discretionary Access Control

With discretionary access control (DAC) the operating system also enforces a set of access rules. The fundamental difference between MAC and DAC is that in a DAC system an object's security attributes may be changed during its lifetime. DAC is discretionary in that the authority for a subject to access an object is left to someone's discretion. In most systems that someone is the user who owns a particular object, however, some systems limit this discretion to some other selected individual. The owner does not directly allocate access rights to subjects. Rather, the owner allocates

access rights to specific user IDs. Thus access to resources lies at the discretion of one or more individuals.

C.   IDENTIFICATION AND AUTHENTICATION

Identification is the process by which an entity informs another entity of their existence. Authentication is the process by which one entity proves to another the authenticity of the identification. In computers, the most common method of identification and authentication is the use of a user ID and a password [GAS88]. The operating system prompts the user to enter his user ID. It then prompts the user to enter his password. The operating system then looks up the password entry for the supplied user ID in an internally maintained database. If the password entry matches the one the user supplied, he has successfully logged onto the system. Thus, the user ID is the identification and the password is the authentication.

Clearly, the previous sections on MAC and DAC illustrate the importance of accurate identification and authentication. Both systems rely on the user ID to set a subject's security attributes. The subject's attributes are then used to determine what actions may be performed. If an individual possesses the user ID and password of a legitimate user, he can use them to log onto the system. As far as the system is concerned, the individual is the legitimate user. After all, he provided the proper identification and authentication. While a user's ID is usually not a secret, his password is. A user must protect his password.

D.   THE TRUSTED PATH

A potential difficulty in protecting a password from disclosure occurs during the very time it must be divulged. During the logon procedure, the user must provide both

3

his user ID and password. As previously stated, the user ID / password pair is how the user identifies and authenticates himself to the operating system. However, has the operating system authenticated itself to the user? While the logon prompts may look like they have been provided by the operating system's logon process, a user process illegitimately masquerading as the logon process may have provided them. This masquerading process can steal passwords. Such a process is an example of a "Trojan horse."

A Trojan horse appears to be one thing when it is actually another [PLF97]. While it may or may not provide its expected services, it also provides some other service that the user does not know about and would not willingly request. A Trojan horse acting as the logon process and recording a user ID and password pair is known as a logon spoofer. A logon spoofer can store the captured information and/or send it to an unauthorized individual. This can result in unauthorized use of the computer system. Some method is required to allow the operating system to authenticate itself to the user. This method is known as a trusted path.

A trusted path is a communication channel between the user and the operating system. It is invoked through some action initiated by the user that is guaranteed to result in communication with the operating system directly. This may take the form of a certain combination of keystrokes, as in the XTS-300 or Microsoft Windows NT/2000, or by placing the mouse on a certain portion of the screen in a graphical user interface, as in Trusted Solaris. The keystroke method of invoking the trusted path is referred to as a secure attention key (SAK), because it is a secure method of requesting the operating

system's attention. On systems that support a trusted path, it must be invoked for actions that require communication with security-critical portions of the operating system. Such actions may include the previously mentioned process of logging on to the system, changing a password, logging off, etc.

In the subsections that follow, the basic security features of several operating systems are summarized. Specific attention is paid to the manner in which a user invokes a trusted path and how the operating system ensures that user processes are not able to interact with the user during the time that he is communicating with security-critical portions of the operating system. Attention is also paid to the manner in which each operating system handles simultaneous users.

1.   Windows NT/2000

Microsoft Windows NT and Microsoft Windows 2000 are two operating systems that enforce a discretionary access control. Both also provide an implementation of a trusted path. The SAK for these systems is the pressing of the Ctrl-Alt-Del keys simultaneously. This invokes a trusted path to the logon process. Only after these keys are pressed does the logon dialog box appear. The user then enters his user ID and password and is validated by the system. Note that in Windows 2000 Professional the requirement to press the SAK in order to logon is not enabled by default. It is enabled by default for the other products, e.g. NT Workstation, NT Server, and 2000 Server.

Once a user is logged onto the system, security critical communication with the operating system is accomplished by pressing the SAK. This displays the "Windows

Security" dialog box [MIC00]. This consists of an information section which displays the

- User ID

- Logon date and time

It also consists of a panel containing the following buttons:

- Lock Computer – locks the console until the SAK is again pressed and the user reenters his password.

- Log Off – logs the user off of the system.

- Shut Down – logs the user off of the system and causes the operating system to stop all programs, in an orderly manner, so the computer may be turned off.

- Change Password – prompts the user to reenter his old password, for confirmation, and type in, twice, a new password.

- Task Manager – brings up the task manager. The task manager allows the user to kill applications or processes and to view system performance information.

- Cancel – returns the user to whatever he was previously doing.

By requiring a user to press the SAK prior to logging on or changing his password, Windows NT/2000 ensures that a logon-spoofing program left by another user is not attempting to capture his password. The other actions (i.e., log off, shut down, and displaying the task manager) can all be initiated through other means in addition to pressing the SAK.

While multiple users may be simultaneously logged on to a Windows NT/2000 system, only one user at a time may be logged onto the computer from the physical console. The other users must logon to the system via a network connection. For the console user, the system ensures that pressing the SAK temporarily halts all screen output

6

of running processes during the time that the user is interacting with the trusted path. Otherwise, a previous user could leave a trusted path window masquerade program executing.

2. Trusted Solaris

Sun's Trusted Solaris 7 provides an implementation of a trusted path on an operating system that enforces mandatory access control and discretionary access control. Sun has chosen not to require the pressing of a set of keys to invoke a trusted path as part of the logon process. All that is required to logon is for the user to enter his user ID in one dialog box and then enter his password in another. This method is secure due to the trusted path that is present when the user is logged on. When a user is logged on, the bottom of the screen has an area known as the "trusted stripe." It displays security information, such as the security level of the currently selected window. Whenever the mouse cursor is positioned in an area of the graphical user interface that has direct communication with the operating system, a trusted path indicator appears at the left end of the trusted stripe. Trusted Solaris stops user level processes from drawing over the trusted stripe. Therefore, there is no need for a SAK; the user invokes the trusted path via mouse placement. The existence of the trusted stripe is the reason that a user process cannot spoof the login dialogs. When logging-in, it is the absence of the trusted stripe that informs the user that he is being presented with the authentic logon dialogs. As cautioned in the Trusted Solaris User's Guide:

> You should *never* see the Trusted Stripe when the login screen appears. If you ever see the screen stripe while attempting to log in or unlock the screen, do not type your password because there's a chance you

7

are being spoofed, that is, an intruder's program is masquerading as a login program to capture passwords. [SUN99]

Once a user is logged onto the system, the standard list of trusted path options is presented when the user clicks the right mouse button in the center of the panel just above the trusted stripe. Among the options is one to allow the user to change his password. See Figure 1. Once again, this must be done via a trusted interface to ensure a user level process is not spoofing the "change password" process.

Since Trusted Solaris implements mandatory access controls, there are additional options, both during login and via the trusted path menu to set security attributes. During logon a menu appears allowing the user to select the range of attributes to allow during this logon. The range must fall within the limits set by the administrator for the user.

Figure 1. Trusted Path Menu [SUN99]

Once the user is logged-on, he may specify the security attributes for future windows that he opens. This allows the user to have multiple windows open

simultaneously, each with the same or different security settings. However, just like Windows NT/2000, only one user can be logged onto the system from the console at any one time.

3.   STOP

Wang Federal's STOP is a secure operating system developed for the XTS-300. It provides an implementation of a trusted path on an operating system that enforces mandatory access control and discretionary access control. STOP is a custom-built operating system that provides an emulation of the Unix operating system. Unlike either Windows NT/2000 or Trusted Solaris, STOP has a text-based trusted path user interface. A windowing system, X Window, can be run on top of the operating system, however, interactions with the trusted elements of STOP are accomplished with a text-based interface.

For this operating system the SAK is invoked by simultaneously pressing the Alt-SysRq keys. The system then displays a welcome message, and prompts the user for a user ID and password. Once authenticated, the user is informed of the current security attributes and is presented with a standard Unix command prompt. If the user wishes to invoke the trusted path the SAK is pressed. This clears the screen and presents the user with the trusted menu shown in Figure 2.

```
Entering trusted environment at sec lvl 0 int lvl 3
Detached from process family 1
Enter command
?
```

Figure 2. STOP Trusted Menu

Once in the trusted environment, the user can enter any of the following

commands [WAN98]:

- **cdl** – change default level – changes the default security level for the user's future logons

- **chd** – change home directory

- **cup** – change user password

- **disconnect** – allows the user to disassociate a specified process family from a user session. This allows processes to continue running even if the user logs out of the system.

- **fsm** – file system management – allows the user to perform various file system commands, i.e. change directory, copy file, etc.

- **ikill** – immediate kill – immediately terminates all processes associated with a process family.

- **kill** – initiates termination of all processes associated with a process family. The processes are sent a signal informing them that the yare being killed. However there is no guarantee that the processes will comply.

- **logout** – kills all processes associated with the user that have not been disconnected and logs the user out of the system.

- **reattach** – reattaches the user to a process family.

- **run** – begins execution of a process family.

- **session** – displays the status of the current session.

- **sg** – set group – sets the group identifier for the current session.

- **sl** – set level – allows the user to specify the current security attributes

- **system** – displays the status of the system

STOP allows a user to have more than one session open. Each session can have

security attributes set to a valid value within the user's allowed range. STOP refers to

each session as a process family. The user can press the SAK, use the **sl** command to

10

change the security level, and execute the **run** command to start running the new session. This process of switching between sessions is more difficult than that provided by Trusted Solaris where all that is required is clicking the mouse in a window running the desired session.

E.   SESSION CONTROL

If an operating system allows multiple user sessions, it must have some method of controlling their execution. The XTS-300, with its text-based trusted path interface, adopts a technique of completely stopping the execution of a user session. This occurs when the session is not "connected" to the console. A specific session is not connected to the console when:

- The user is interfacing with the trusted path.

- Another session is connected to the console.

For this technique to work there must be some mechanism enabling the operating system to place a process in a "secure halt" state. The operating system can then guarantee that secure halted process will not execute and cannot perform any actions, including malicious ones.

F.   SUMMARY

The remainder of this thesis proposes modifications to the Linux operating system that add the ability to secure halt processes and sessions. This will lay the foundation for a trusted path that is capable of managing multiple sessions. Note that the resulting system, while adding some of the functionality found in the XTS-300, in no way rises to the same level of assurance. To do so would require designing an operating system from the ground up, with special emphasis on security.

The remainder of this thesis is organized as follows:

- Chapter II is a high-level design of the secure halt mechanism design.

- Chapter III is description of the changes required to the existing Linux kernel in order to add a secure halt mechanism.

- Chapter IV provides a summary of the thesis work, including implementation progress, problems encountered, and suggested future research topics.

- Appendix A provides detailed specifications of the new modules.

- Appendix B provides a listing of the source code for new modules.

- Appendix C provides a description of the framework developed to allow testing of the secure halt mechanism.

II.   PROCESS SECURE HALT MECHANISM DESIGN

This chapter describes the high-level design of the secure halt mechanism.

Chapter III describes the specific changes required to Linux in order to implement this

design.

A.   PROCESS STATE MODIFICATIONS

A process is said to be "secure halted" when it will not execute regardless of other

system events. An explicit request to "unhalt" the process is the only way that a process

will return to the normal flow of execution. To understand what is required to achieve

the secure halt status, it is necessary to understand the lifecycle of a process.

1.   Standard Process Lifecycle

The standard lifecycle of a process is shown in Figure 3.
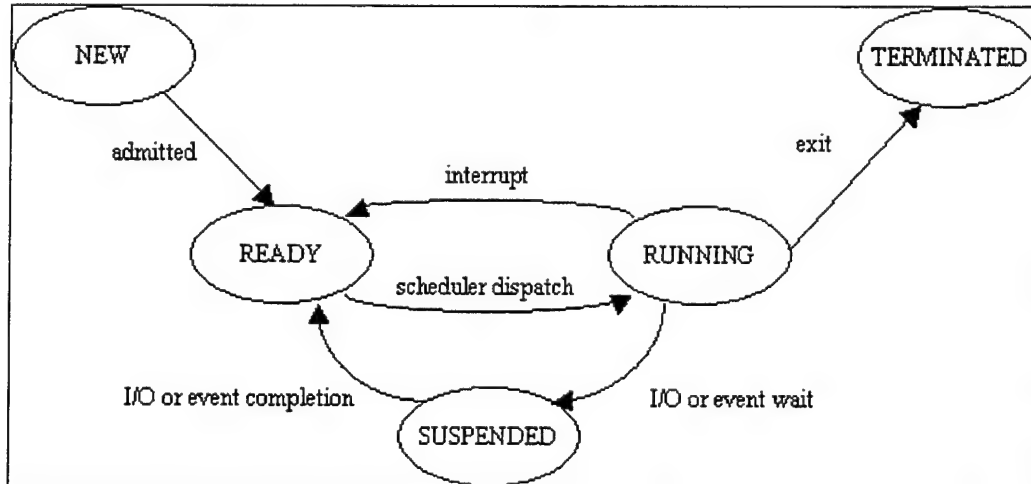
Figure 3. Typical Process State Diagram [SIL98]

The states are described as follows:

• New – A process that is being created.

13

- Ready – A process that is ready to run. It is waiting for the scheduler to assign it to a processor.

- Running – A process that is executing on a processor. This is the only state in which a process is actually "doing" something.

- Suspended – A process is waiting on some input/output or event. This could be keyboard input from the user, a disk read, etc.

- Terminated – A process that has completed its execution. A process will remain in the terminated state until the operating system has determined that the process' information is no longer needed

Adding a secure halt mechanism involves the addition of a new state. This is called the secure halt state.

2. Fully Connected Secure Halt State Process Lifecycle

Adding a new state raises the question: What events will cause the process to enter and leave the secure halt state? Additionally, what current states will a process be allowed to depart in order to enter the secure halt state, and vice versa? Clearly, a process in the new or terminated states will not need to enter the secure halt state. In the first case, the process is being created with the intent of letting it execute. In the second case, the process is finished and will not have a chance to execute again. A possible solution is shown in Figure 4. This solution shows that a process can enter from or depart to the secure halt state from any of the remaining states: ready, running, or suspended. This is called a *fully connected* secure halt state.

There is a problem with this solution. A process in the suspended state is waiting for an I/O operation to complete or waiting for some specified event. If a process is removed from the suspended state and placed in the secure halt state, then it must also be removed from the queue in which it was waiting. What should then be done when the I/O event completes? Does the next process in the queue get the event? Is the event discarded? In either case the event will be lost to the process. What should be done when the process departs the secure halt state and is returned to the suspended state? Is it requeued for the I/O event? Determining how to handle this would be difficult.
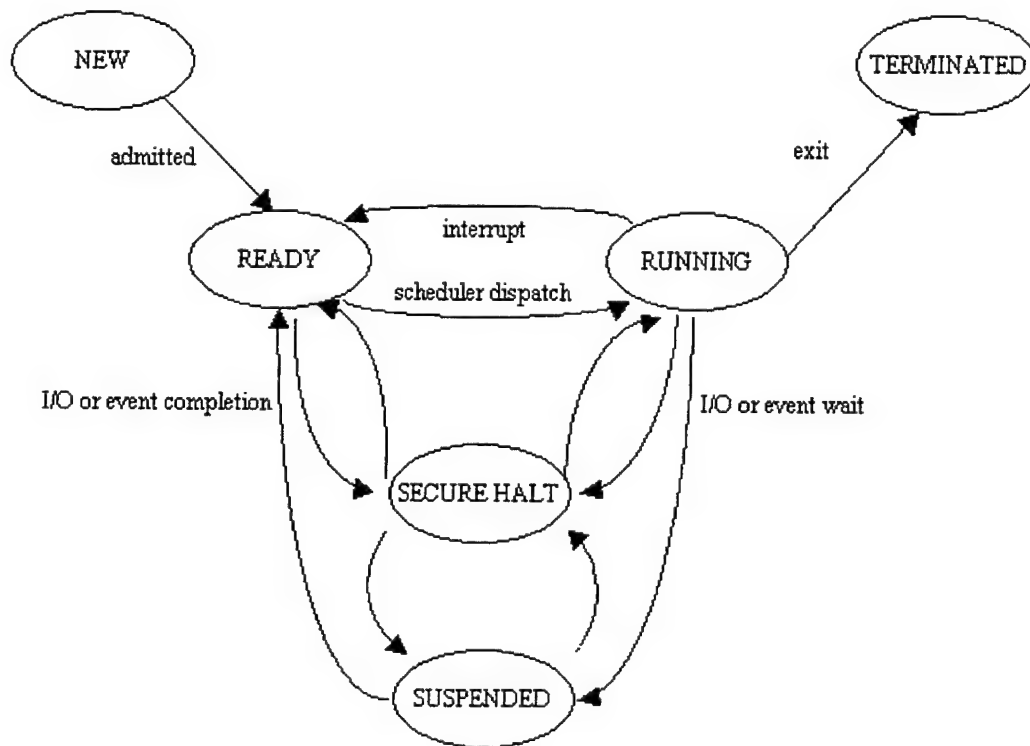
Figure 4. Fully Connected Secure Halt Entry

15

A process in the ready state, and by extension in the ready queue, could easily be removed from and replaced into the ready state and queue. A process in the running state could be removed from the processor, enter the secure halt state, and the scheduler allowed to select a new process from the ready queue to move to the running state. Difficulties may arise, however, when several processes that were formerly running are removed from the secure halt state simultaneously. A decision must be made as to which one will be selected to run, while the others are sent to the ready queue. This decision could be made by the scheduler since this is exactly the type of decision that the scheduler is designed to make. However this does not address the problem of returning a process to the suspended state. These problems render this *fully connected* solution unacceptable.

3.   Single Entry Secure Halt State Process Lifecycle

A second solution is shown in Figure 5.
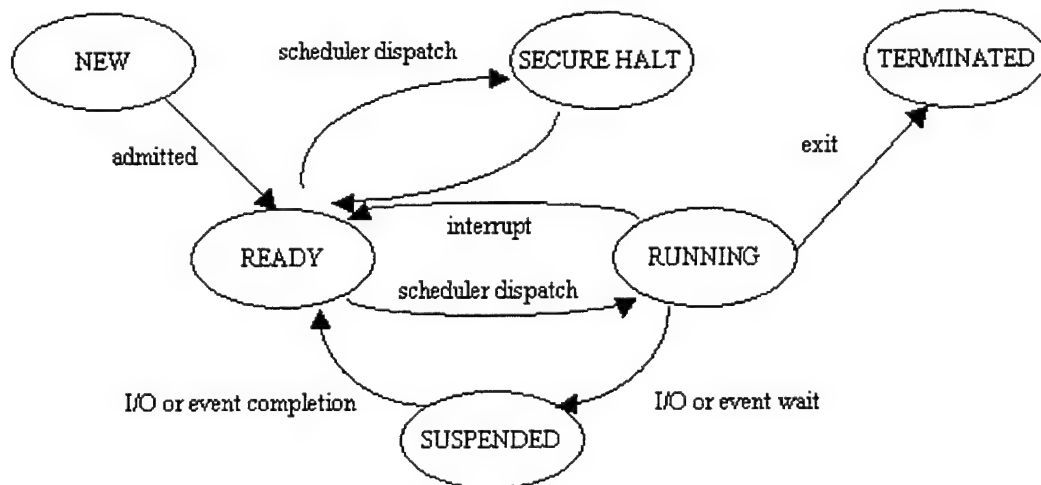
Figure 5. Single Entry/Exit Secure Halt State

16

This solution allows entry to the secure halt state from the ready state, and departure from the secure halt state to the ready state. This is accomplished by having a secure halt router to decide which processes to route to the secure halt state. The secure halt router will send processes that are not secure halted on to the scheduler. Since the scheduler must analyze each process while making its scheduling decisions, the additional check does not change the locus of decision-making. The scheduler normally acts as a "gatekeeper" to enter the running state. The secure halt router acts as a gatekeeper to the scheduler.

This solution implies that a request to enter the secure halt state will not be immediately granted. It will be deferred until the process comes up for scheduling. Enabling this deferral requires a secure halt flag. The flag will be set when a request is received to secure halt the process. The secure halt router, seeing a set flag, will send the process to the secure halt state. If the flag is not set the secure halt router will send the process to the scheduler. When a request is received to remove a process from the secure halt state, the secure halt flag is cleared and the process is returned to the ready state and queue.

This solution solves the problem of the suspended state. A request to enter the secure halt state for a process in the suspended state will result in the secure halt flag being set. The process will continue to wait until the I/O or event is complete. It will then take its normal path to the ready state and queue, where the secure halt router will perform as explained above. A critical issue is ensuring that I/O destined for a process that has been secure halted, will not be sent to another process instead. This would create

17

confusion as well as a major security concern. Since the process has waited for its I/O event, it will receive the I/O data, not some other process.

It raises a new problem: what to do with a process in the running state. When a secure halt request is received, the process cannot be allowed to continue executing. Setting its secure halt flag and immediately invoking the scheduler solves this. The currently running process will then take its normal path to the ready state and queue, where the secure halt router will perform as explained above. Most modern operating systems, such as Windows NT [SOL98] and Linux [BEC98], have an idle process that executes when no other processes are ready. Thus we are guaranteed that the scheduler can find a process that has not been secure halted.

B.   SECURE HALT FLAG

As mentioned in the section on Single Entry Secure Halt State Process Lifecycle, a system that supports a secure halt state will require the addition of a secure halt flag to each process control block. When a process is in the new state, i.e. it is being created, the secure halt flag is initialized to "cleared." When it is requested that a process enter the secure halt state, the secure halt flag must be set. A new code module, the Secure_Halt_Routing_Module must guarantee that it will send to the secure halt state any process it finds in the ready queue with its secure halt flag set. When it is requested that a process depart the secure halt state, its secure halt flag must be cleared prior to its return to the ready queue. Another new module, the Secure_Halt_Module is the part of the operating system responsible setting and clearing the secure halt flag.

18

C. SECURE HALT MECHANISM MODULES

This section describes the modules required to support the process secure halt mechanism. In this context, the term "module" refers to an active part of the system that manages a particular database or flow of control.

1. Secure_Halt_Module

The Secure_Halt_Module is responsible for controlling a process' entry and exit from the secure halt state. It provides a single point of entry for interfacing with the secure halt mechanism. The Secure_Halt_Module will provide the following interface:

- A function for initializing a process' secure halt state.

- A function that returns whether or not a process is in the secure halt state.

- A function for setting a process into the secure halt state. This function must also determine if the process is currently executing. If so it must call the scheduler to remove the process from execution.

- A function for removing a process from the secure halt state. This function must determine if the process is in the secure halt state. If so, it must send the process to the ready state and queue.

2. Secure_Halt_Session_Module

A session is defined as a group of processes that will be managed together. The Secure_Halt_Session_Module is responsible for controlling a session's entry and exit from the secure halt state. It provides a single point of entry for sessions interfacing with the secure halt mechanism. This is equivalent to a process family on an XTS-300 system

or the processes assigned to a virtual terminal in Linux. The

Secure_Halt_Session_Module will provide the following interface:

- A function for querying if all the processes in a session have been secure halted.

- A function for secure halting a session. This function must guarantee that is sets the secure halt flag of each member of the session to be halted. Furthermore, is must guarantee that a currently executing process that is part of the session will be immediately removed from the running state. If so it must call the scheduler to remove the process from execution.

- A function for secure unhalting a session.

3. Secure_Halt_Routing_Module

The Secure_Halt_Routing_Module is responsible for ensuring that a process with its secure halt flag set that is about to be examined by the scheduler, will instead be sent to the secure halt state. A process with its secure halt flag cleared will be sent to the scheduler for normal processing.

III. LINUX MODIFICATIONS

This chapter describes the modifications that must be made to the Linux kernel to support the design proposed in Chapter II.

A. PROCESS STATES IN LINUX

Once again the starting point will be analyzing process states, this time in a Linux system, as shown in Figure 6.
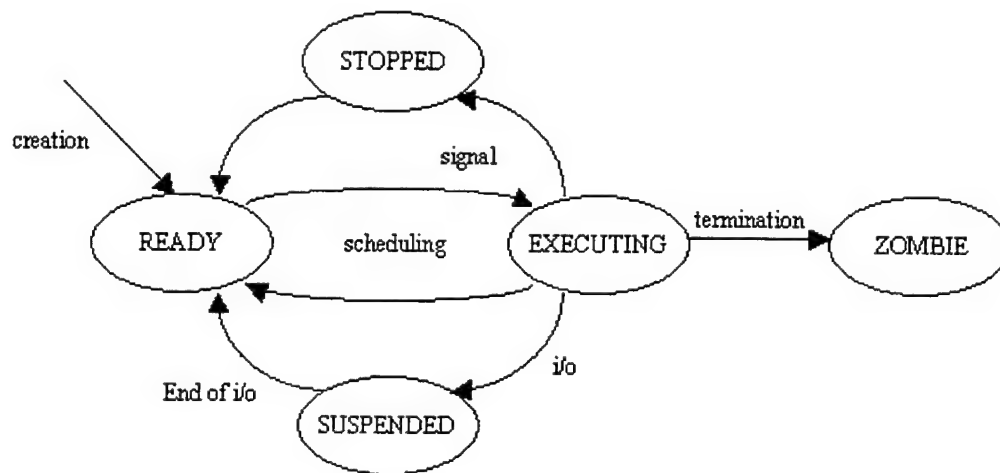
Figure 6. Linux Process State Diagram [CAR98]

These states are defined as follows:

- Ready – A process that is ready to run. It is waiting for the scheduler to assign it to a processor.

- Executing – A process that is executing on a processor. This is the only state in which a process is actually "doing" something.

- Suspended – A process is waiting on some input/output or event. This could be keyboard input from the user, a disk read, etc.

21

- Zombie – A process that has completed its execution. A process will remain in the terminated state until the operating system has determined that the process' information is no longer needed

- Stopped – A process has been halted by some other process [CAR98]. The other process can restart the halted process as necessary. The stopped state is usually used by a debugger as it is stepping through a program.

1. Comparison of Standard Process States with Linux' Implementation

A comparison of Figure 6 with Figure 3 is shown in Table 1.

Standard Process State	Linux Process State
New	
Ready	Ready
Running	Executing
Suspended	Suspended
Terminated	Zombie
	Stopped

Table 1. Generic O/S vs. Linux Process States

While Linux does not have a "new" state, it does recognize that a process in the midst of creation does not fall into one of the other states. Of note is Linux' addition of a stopped state. In the stopped state, a process is unable to proceed until restarted by a controlling process. Since the controlling element is a process, and therefore, in user space, the stopped state is not acceptable as a substitute for our concept of a secure halt state. Control of entry to and exit from the secure halt state must reside in the kernel.

Linux cannot, by any stretch of the imagination, be considered a secure operating system. Nevertheless, potentially allowing control of the secure halt mechanism from outside the kernel would not be a good design decision. A better design would be to have a new method of implementing a secure halt mechanism that depends on a minimal amount of current mechanisms [SCH75].

Linux further subdivides the waiting state into two sub states: interruptible and non-interruptible [MAX99]. In the interruptible state, the process is waiting for an event. However, sending it a signal may wake it up. Signals are the Linux implementation of inter-process communication [BEC98]. There are signals to kill a process, to wakeup a process, etc. In the uninterruptible state, the waiting process may not be woken up by a software signal. It will remain in the waiting state until some hardware condition occurs.

2. Actual Linux Kernel Implementation of Process States

The Linux kernel defines a process through a **task_struct** defined in the file "include/sched.h" [TOR92b]. Each process, upon creation, has a **task_struct** allocated to it. One of the members of the **task_struct** is the **state** variable. The state variable, which is a long integer, is supposed to be assigned one of the defined process states. The Linux kernel defines process states in the header file, "include/sched.h" [TOR92b]. These defined states are:

- TASK_RUNNING

- TASK_INTERRUPTIBLE

- TASK_UNINTERRUPTIBLE

- TASK_ZOMBIE

- TASK_STOPPED

- TASK_SWAPPING – this state is never assigned to the state variable.

The above list does not include something like "TASK_READY." This is because all tasks that are ready to run are assigned the TASK_RUNNING state. They are all placed in the **ready_queue**. When the scheduler selects a process to run on a processor, it is not removed from the **ready_queue**. Nor is its **state** variable changed from TASK_RUNNING. Instead, its **has_cpu** flag, also declared in the **task_struct**, is set. The net result is that a process is in the ready state if and only if the following conditions are true:

- Its **state** variable is set to TASK_RUNNING.

- It is in the **ready_queue**.

- Its **has_cpu** flag is cleared.

A process is in the running state if and only if:

- Its **state** variable is set to TASK_RUNNING.

- It is in the **ready_queue**.

- Its **has_cpu** flag is set.

Thus, although a process' state variable is never set to some "ready" value, the Linux kernel does have a ready state, albeit a virtual one.

3.   Linux Kernel Process State References

A search of the kernel source using the Linux Cross Reference [GLE00] gave results shown in Table 2.

24

Process State	Number of Files Referencing
TASK_RUNNING	114
TASK_INTERRUPTIBLE	186
TASK_UNINTERRUPTIBLE	51
TASK_ZOMBIE	14
TASK_STOPPED	26
TASK_SWAPPING	3

Table 2. Process State Access Search Results

Scrutinizing some of the actual references shows that many of them involve the modification of a process' **state** variable. This is a gross violation of the philosophy of limiting changes to a few, well-specified modules. Attempting to implement the secure halt state as a new legal value to assign to a process' **state** variable would not be very secure. This leaves the following problem: how to implement the secure halt state in code?

B. ADDING THE SECURE HALT FLAG AND STATE

1. Adding the Secure Halt Flag

Recall that the secure halt flag signifies that a process must not be placed in the running state, and that it must be diverted to the secure halt state by the secure halt routing module prior to reaching the scheduler. As stated in the high-level design in Chapter II, the secure halt flag is used to notify the secure halt routing module that a process in the **ready_queue** must be sent to the secure halt state. The simplest location

25

to place the flag, which is required for each process, is in the **task_struct** assigned to each process. Therefore, the secure halt flag was added to the task_struct. Also, code was added to the initialization of the **task_struct** so that a process is created with its secure halt flag cleared.

2. Adding the Secure Halt State

Returning to the problem of implementing a secure halt state, a solution presents itself that is an extension of the manner in which Linux implements the ready and running states. A process can be said to be in the secure halt state if and only if all of the following conditions are true:

- Its state variable is set to TASK_RUNNING.

- It is in the **ready_queue**.

- Its **has_cpu** flag is cleared.

- Its **secure halt** flag is set.

This approach has an important benefit. It makes use of a new addition to a process' **task_struct**: the **secure halt** flag. Since this new flag is unknown to the rest of the kernel, a well-defined interface, the secure halt module, can be used to centralize access to the flag.

C. SECURE HALT MECHANISM MODULES

This section describes the modules required to support the secure halt mechanism. Critical to the module implementation is an understanding of the potential for the Linux kernel to be multi-threaded. One of the choices to make when compiling the Linux kernel is whether or not to enable symmetric multi-processing (SMP) support. SMP

support allows the kernel to make full use of multiple processors on SMP-compliant hardware. However, this means that the kernel's data structures, including the process' **task_struct**, and the **ready_queue**, must be protected from simultaneous attempts at modification. This is accomplished through the classical use of locks and semaphores. A primary consideration in designing these modules is that they each specialize in one area of the secure halt mechanism [PAR72].

1.   Secure Halt Module

The secure halt module provides functions for:

- Querying a process' secure halt status.

- Placing a process in the secure halt state.

- Removing a process from the secure halt state.

The second and third functions modify the **secure halt** flag in a process' **task_struct**. Therefore, they must properly lock the **task_list** while making their modifications. The **task_list** is a list containing all tasks in the system, regardless of their current state. These functions must also ensure that they release their lock on the **task_list** when completed.

Also critical is the need to deal with a request to secure halt a process that is running. This requires, in addition to the standard behavior, that the function call the scheduler to immediately cause it to perform scheduling. This will ensure that the process is removed from the running state.

2. Secure Halt Session Module

While the secure halt module is responsible for managing the secure halt status of individual processes, the secure halt session module is responsible for managing the secure halt status of entire sessions. Linux supports the concept of virtual terminals. A typical Linux system has six virtual terminals. Each virtual terminal has the ability to manage one user login. Thus in the typical system six users, or the same user six times, or any combination, can be logged-in to a Linux system from one keyboard and monitor. Each of these virtual terminals has a unique session ID associated with it. Furthermore, each process created in support of a virtual terminal has the **session** variable of its **task_struct** set to the virtual terminal's session ID.

The secure halt session module provides functions for:

- Querying a session's secure halt status.

- Placing a session in the secure halt state.

- Removing a session from the secure halt state.

Note that a session is in the secure halt state if and only if all processes that are part of that session are in the secure halt state.

The most straightforward way to implement the functions of the secure halt session module would be to iterate through the **task_list** and call the matching function in the secure halt module for each process that is part of the session in question. However, this is not the most efficient approach. Recall that the secure halt modules modification functions lock and unlock the **task_list** while performing their work. For a session

containing multiple processes, this would result in multiple attempts to lock and unlock the **task_list**.

The approach actually implemented is to have the secure halt session module's modification functions lock the **task_list**, iterate through the **task_list**, modify the secure halt flag for processes that are part of the session in question, and release the lock on the **task_list**. While this approach improves efficiency, it does so at the cost of a layered design. Once again special attention was paid to ensure that if a process that is part of a session being secure halted is running, that the scheduler will be called to remove it from the running state.

3. Secure Halt Routing Module

Implementation of the secure halt routing module requires an understanding of the Linux scheduler.

a. The Linux Scheduler

The Linux scheduler selects a process in the ready state and places it in the running state. The actual algorithm used by the scheduler can be changed and is irrelevant to the current discussion. The scheduler iterates through the ready queue searching for processes that are schedulable. The definition of a schedulable process depends on whether the kernel is compiled with or without SMP support.

If the kernel is compiled without SMP support, then a schedulable process is defined as any process in the **ready_queue**. Even if a process is currently on the one, and only processor, it can potentially be selected for continued execution. If the kernel is compiled with SMP support, then a schedulable process is defined as any process in the

29

ready_queue that is not currently running on another processor. It would not be efficient to force a process off of one processor because another one has selected it.

Once the scheduler has determined that a process is schedulable, it computes a "goodness" value. The goodness value represents a way of deciding which process is the most suitable for selection to run. If the process currently being examined by the scheduler has a higher goodness than previously seen, it is selected as the potential winner of the selection process. This goes on until the scheduler has examined all processes in the ready queue. The scheduler then declares the potential winner the actual winner and proceeds to place it into the running state.

b. Adding The Secure Halt Routing Module

Given the proposed implementation of the secure halt flag and state, implementation of the secure halt routing module is straightforward. Recall the secure halt router's task: ensure that the scheduler cannot place a process in the secure halt state into the running state. Decision-making by the secure halt routing module can take place within the scheduler. If the module's decision-making code is collocated with the scheduler's determination of a schedulable process, then it can properly ensure that the scheduler cannot select a securely halted process for execution.

The decision-making code consists of calling the secure halt module's query function for the process being tested for schedulability. If the process has been secure halted, then it will not be analyzed by the scheduler and, therefore, will not be selected for execution.

IV. CONCLUSIONS

The goal of this research was to add a mechanism allowing the Linux operating system to halt processes in a security-related manner. This mechanism provides key functionality necessary for future design and implementation of a trusted path. This chapter details the progress made toward this goal, the problems encountered, and areas of future research.

A. PROGRESS MADE

The secure halt mechanism was successfully added to the Linux operating system. The SECURE_HALT_MODULE and the SECURE_HALT_SESSION_MODULE were both fully implemented. A testing framework, as described in Appendix C, was designed to test the efficacy of the design. It was confirmed that all processes that were part of a session that had been halted were no longer able to execute until subsequently unhalted.

B. PROBLEMS ENCOUNTERED

The primary problem was the need to understand how the Linux kernel handled the scheduling of processes and how it defined process states. Unfortunately, the Linux kernel makes heavy use of "goto" statements and global variables. These make understanding the logical flow difficult. The most useful tool for finding the declaration and use of variables was the Linux Cross Reference [GLE00].

Additionally, the usual difficulties were encountered when developing at the operating system level. While compiling the kernel source code is straightforward, it requires several steps that must be carefully followed [WAR01]. The complete turn-around time for making a small change to the kernel, compiling, rebooting, testing

31

the new kernel, and rebooting back to the development environment took an average of 20 minutes. Furthermore, as a debugger cannot be used on the kernel, it was necessary to use the kernel printing facility, **printk**, to generate debugging traces and view variable values. This was not a very efficient procedure, but it was all that was available.

One helpful configuration was to have the development Linux installation share a common boot partition with the secure halt mechanism test Linux installation. This greatly simplified installation of the newly compiled kernel.

It may be possible to reduce the turn-around time by developing on a computer running VMware. VMware is a software program that sits between the hardware and the operating system. It allows multiple virtual machines, i.e. installations of an operating system, to run concurrently. Each machine thinks it is running directly on the hardware. Developing on such a system would allow development on one virtual machine and testing on another. The testing machine could be rebooted without disturbing the development machine. [VMW01]

An examination was made into the feasibility of modifying the **ps** command so it will display the secure halt status of processes (see Future Research). Several attempts were made to compile the original source code, without modification [JOH00]. These attempts were unsuccessful and the examination was terminated.

C. FUTURE RESEARCH

1. Trusted Path

The secure halt mechanism was developed as a support mechanism for adding a trusted path to Linux. The difficulty in adding a trusted path will center on performing

I/O from within the kernel. Currently, the kernel print command, **printk**, can be used to display output to the user from within the kernel. No comparable command exists for accepting input from the user. A possible solution may be found in capturing all keystrokes at the keyboard driver level when the trusted path has been invoked. Solving the input problem would go a long way toward designing a functional trusted path for Linux.

2. Disabling I/O Buffering

When a session is suspended, even though none of the processes belonging to the session can execute, the keyboard I/O queue continues to buffer keystrokes. All of the keystrokes entered for a session while the session is secure halted are buffered and given to the session when it is unhalted. While this is not as dangerous as if the keystrokes were being given to an incorrect process, it is not a desired behavior. Research should be done into how the keyboard I/O queue buffers keystrokes and how to stop this from happening for a session while it is secure halted.

3. ps Command

The **ps** command, which displays process status information, should be modified so it is able to display the secure halt status of each process. Given that the secure halt mechanism has been added in kernel-space and the **ps** command executes in user-space, enabling this functionality will require determining how to export the secure halt state information to a location accessible to the **ps** command and importing the data into the **ps** command. A cursory examination of the source code showed that the **ps** command gathers it information by parsing information stored in the /proc directory for each

33

process. Logically, adding the secure halt state information to the data written by the kernel into the /proc directory should go a long way to adding this functionality.

APPENDIX A. MODULE DESIGN

A.   SECURE HALT COMMON TYPES MODULE

This module defines constants used by other modules

External Constants:

- SEC_HALT_TRUE

- SEC_HALT_FALSE

- SEC_HALT_SUCCESS

B.   SECURE_HALT_MODULE

This module defines the interface of the Secure_Halt_Module described in

Chapter III. This module does not depend on any other modules.

External Constants:

- SEC_HALT_ERROR_INVALID_PROCESS

External Entry Points

- init_secure_halt

- is_secure_halted

- secure_halt_task

- secure_unhalt_task

1.   init_secure_halt

a.   Processing

Sets p's secure_halted flag so that a call to is_secure_halted will return

SEC_HALT_FALSE.

b.   *External Interface*

```
int init_secure_halt(
        task_struct *p
);
```

c.   *Inputs*

- p – the process which needs its secure halt flag initialized.

d.   *Outputs*

- <function result>

    o   SEC_HALT_SUCCESS if p's secure_halted flag was

        properly initialized.

    o   SEC_HALT_ERROR_INVALID_PROCESS if p is

        NULL.

2.   is_secure_halted

a.   *Processing*

Checks the status of the secure_halt flag.

b.   *External Interface*

```
int is_secure_halted(
        task_struct *p
);
```

c.   *Inputs*

- p – the process whose secure halt flag is being questioned.

d. *Outputs*

- \<function result\>
    - o SEC_HALT_TRUE if p's secure_halted flag is set.
    - o SEC_HALT_FALSE if p's secure_halted flag is not set.
    - o SEC_HALT_ERROR_INVALID_PROCESS if p is NULL.

3. secure_halt_task

a. *Processing*

Sets p's secure_halted flag so that a call to is_secure_halted will return SEC_HALT_TRUE. If p is currently executing, this function calls the scheduler to remove p from execution.

b. *External Interface*

```
int secure_halt_task(
        task_struct *p
);
```

c. *Inputs*

- p – the process which is to have its secure_halted flag set.

d. *Outputs*

- \<function result\>
    - o SEC_HALT_SUCCESS if p's secure_halted flag was successfully set.

o SEC_HALT_ERROR_INVALID_PROCESS if p is

NULL.

4. secure_unhalt_task

a. *Processing*

Sets p's secure_halted flag so that a call to is_secure_halted will return

SEC_HALT_FALSE. Also, the process is returned to the ready state and queue.

b. *External Interface*

```
int secure_unhalt_task(
        task_struct *p
);
```

c. *Inputs*

- p – the process which is to have its secure_halted flag cleared.

d. *Outputs*

- <function result>

    o SEC_HALT_SUCCESS if p's secure_halted flag was

    properly cleared.

    o SEC_HALT_ERROR_INVALID_PROCESS if p is

    NULL.

C. SECURE_HALT_SESSION_MODULE

This module defines the interface of the Secure_Halt_Session_Module described

in Chapter III. This module depends on the Secure_Halt_Module and the

Scheduler_Module.

External Constants:

- SEC_HALT_ERROR_INVALID_SESSION

External Entry Points

- is_secure_halted_session

- secure_halt_session

- secure_unhalt_session

1.　is_secure_halted_session

a.　*Processing*

Iterates through the task_list. For each task, p, whose session number matches the input parameter, it checks is_secure_halted(p). If all such processes return SEC_HALT_TRUE, this function returns SEC_HALT_TRUE. If at least one of the processes returns SEC_HALT_FALSE, this function returns SEC_HALT_FALSE. If no process in the task list has a session number matching the input parameter, this function returns SEC_HALT_ERROR_INVALID_SESSION.

b.　*External Interface*

```
int is_secure_halted_session(
        int session
);
```

c.　*Inputs*

- session – the session whose processes are being queried as to their secure_halted flag status.

d. *Outputs*

- \<function result\>

    o SEC_HALT_TRUE if every process in the input session has its secure_halted flag set.

    o SEC_HALT_FALSE if not every process in the input session has its secure_halted flag set.

    o SEC_HALT_ERROR_INVALID_SESSION if the input session is not a valid session ID.

2. secure_halt_session

a. *Processing*

Iterates through the task_list. For each task, p, whose session number matches the input parameter, it calls secure_halt_task(p). If any of the tasks is currently executing on a processor, the Scheduler_Module's schedule() function is called to immediately perform scheduling. If no process in the task list has a session number matching the input parameter, this function returns SEC_HALT_ERROR_INVALID_SESSION. Otherwise this function returns SEC_HALT_SUCCESS.

b. *External Interface*

```
int secure_halt_session(
        int session
);
```

c.   *Inputs*

- session – the session whose processes are having their secure_halted flag set.

d.   *Outputs*

- \<function result\>

  o   SEC_HALT_SUCCESS if every process in the input session has its secure_halted flag successfully set.

  o   SEC_HALT_ERROR_INVALID_SESSION if the input session is not a valid session ID.

3.   secure_unhalt_session

a.   *Processing*

Iterates through the task_list. For each task, p, whose session number matches the input parameter, it calls secure_unhalt_task(p). If no process in the task list has a session number matching the input parameter, this function returns SEC_HALT_ERROR_INVALID_SESSION. Otherwise this function returns SEC_HALT_SUCCESS.

b.   *External Interface*

```
int secure_unhalt_session(
        int session
);
```

41

Inputs

- session – the session whose processes are having their

  secure_halted flag cleared.

d. *Outputs*

- <function result>

  o SEC_HALT_SUCCESS if every process in session has its

    secure_halted flag successfully cleared.

  o SEC_HALT_ERROR_INVALID_SESSION if session is

    not a valid session ID.

D. SECURE_HALT_ROUTING_MODULE

This module defines the interface of the Secure_Halt_Routing_Module described

in Chapter III.

External Entry Point

- schedulable

1. schedulable

a. *Processing*

Queries the process' is_secure_halted result. If the result is

SEC_HALT_FALSE, the process is not sent to the scheduler. Otherwise, the process is

sent to the scheduler.

b.   *External Interface*

```
int schedulable(
        task_struct *p
);
```

c.   *Inputs*

- p – the process which is to be checked for schedulability.

d.   *Outputs*

- <function result>

    o SEC_HALT_TRUE if p's secure_halted flag is set.

    o SEC_HALT_FALSE if p's secure_halted flag is not set.

    o SEC_HALT_ERROR_INVALID_PROCESS if p is

       NULL.

APPENDIX B. SOURCE CODE

This appendix contains the source code enabling a secure halt mechanism in
Linux. The secure halt mechanism was implemented on version 2.2.18 of the Linux
kernel. Each section is devoted to a module with each subsection corresponding to
individual source and header files.

A.   SECURE HALT COMMON TYPE MODULE

1.   include/linux/secure_halt_types.h

```
/*
 ********************************************************************
 * File: include/linux/secure_halt_types.h
 *
 * Description:
 *     This file contains types and constants that are common to all
 *     Secure Halt modules.
 *
 * 2001-04-05  Created by Jerome Brock.
 ********************************************************************
 */

#ifndef _SECURE_HALT_TYPES_H
#define _SECURE_HALT_TYPES_H


#define SEC_HALT_TRUE   (1)
#define SEC_HALT_FALSE (0)

#define SEC_HALT_SUCCESS (0)


#endif /* defined(_SECURE_HALT_TYPES_H) */
```

B.   SECURE_HALT_MODULE

1.   include/linux/sched.h

- Added

```
#include <linux/secure_halt_types.h>
```

- Added to task_struct

```
struct task_struct {
...
/* secure halt flag */
  int secure_halted:1;
};
```

- Added to INIT_TASK

```
#define INIT_TASK \
...
/* sec halt */  SEC_HALT_FALSE, \
}
```

2.   include/linux/secure_halt.h

```
/*
 ********************************************************************
 * File: include/linux/secure_halt.h
 *
 * Description:
 *     This is the header file for code which places tasks into
 *     and out of the secure halt mode.
 *
 * 2001-04-05  Created by Jerome Brock.
 ********************************************************************
 */

#ifndef _SECURE_HALT_H
#define _SECURE_HALT_H

#include <linux/secure_halt_types.h>
#include <linux/sched.h>

#define SEC_HALT_ERROR_INVALID_PROCESS (1001)

/*
 ********************************************************************
 * Function:
 *     is_secure_halted
 * Inputs:
 *     p          The task whose state is being checked.
 * Outputs:
 *     <result>   SEC_HALT_TRUE if p is secure halted.
 *                SEC_HALT_FALSE if p is not secure halted.
 *                SEC_HALT_ERROR_INVALID_PROCESS if p is NULL.
 * Description:
 *     Returns whether a specified process is secure halted.
 ********************************************************************
 */
extern __inline__ int is_secure_halted(struct task_struct *p){
```

46

```
        if (p == NULL){
                return(SEC_HALT_ERROR_INVALID_PROCESS);
        }
        else if (p->secure_halted == SEC_HALT_FALSE){
                return(SEC_HALT_FALSE);
        }
        else{
                return(SEC_HALT_TRUE);
        }

}

/*
 **********************************************************************
 * Function:
 *     secure_halt_task
 * Inputs:
 *     p           The task which is being secure halted.
 * Outputs:
 *     <result>  SEC_HALT_SUCCESS if p was successfully halted.
 *               SEC_HALT_ERROR_INVALID_PROCESS if p is NULL
 * Description:
 *     Sets a task's secure halt flag.  If the task is
 *     currently executing, this call will call the scheduler, which
 *     will swap out the task.
 **********************************************************************
 */
extern int secure_halt_task(struct task_struct *p);

/*
 **********************************************************************
 * Function:
 *     secure_unhalt_task
 * Inputs:
 *     p           The process which is being secure unhalted.
 * Outputs:
 *     <result>  SEC_HALT_SUCCESS if p was successfully unhalted.
 *               SEC_HALT_ERROR_INVALID_PROCESS if p is NULL
 * Description:
 *     Clears a task's secure halt flag.
 **********************************************************************
 */
extern int secure_unhalt_task(struct task_struct *p);

#endif /* defined(_SECURE_HALT_H) */
```

3.   kernel/secure_halt.c

```
/*
 **********************************************************************
 * File: kernel/secure_halt.c
 *
```

```
 * Description:
 *     This is the implementation file for code which places tasks
 *     into and out of the secure halt mode.
 *
 * 2001-04-05  Created by Jerome Brock.
 *******************************************************************
 */

#include <linux/secure_halt.h>

/*
 *******************************************************************
 * Function:
 *     secure_halt_task
 * Inputs:
 *     p          The task which is being secure halted.
 * Outputs:
 *     <result>   SEC_HALT_SUCCESS if p was successfully halted.
 *                SEC_HALT_ERROR_INVALID_PROCESS if p is NULL
 * Description:
 *     Sets a task's secure halt flag.  If the task is
 *     currently executing, this call will call the scheduler, which
 *     will swap out the task.
 *******************************************************************
 */
int secure_halt_task(struct task_struct *p){
  if (p == NULL){
    return(SEC_HALT_ERROR_INVALID_PROCESS);
  }

  write_lock(&tasklist_lock);
  p->secure_halted = SEC_HALT_TRUE;
  write_unlock(&tasklist_lock);

  if (p == current){
    /* if p is the currently executing process, call the scheduler. */
    schedule();
  }
  return(SEC_HALT_SUCCESS);
}

/*
 *******************************************************************
 * Function:
 *     secure_unhalt_task
 * Inputs:
 *     p          The process which is being secure unhalted.
 * Outputs:
 *     <result>   SEC_HALT_SUCCESS if p was successfully unhalted.
 *                SEC_HALT_ERROR_INVALID_PROCESS if p is NULL
 * Description:
 *     Clears a task's secure halt flag.
 *******************************************************************
 */
```

```
int secure_unhalt_task(struct task_struct *p){
  if (p == NULL){
    return(SEC_HALT_ERROR_INVALID_PROCESS);
  }

  write_lock(&tasklist_lock);
  p->secure_halted = SEC_HALT_FALSE;
  write_unlock(&tasklist_lock);

  return(SEC_HALT_SUCCESS);
}
```

C.   SECURE_HALT_SESSION_MODULE

1.   include/linux/secure_halt_session.h

```
/*
 ********************************************************************
 * File: include/linux/secure_halt_session.h
 *
 * Description:
 *    This is the header file for code which places sessions into
 *    and out of the secure halt mode.
 *
 * 2001-04-05   Created by Jerome Brock.
 ********************************************************************
 */

#ifndef _SECURE_HALT_SESSION_H
#define _SECURE_HALT_SESSION_H

#define SEC_HALT_ERROR_INVALID_SESSION (2001)

/*
 ********************************************************************
 * Function:
 *    is_secure_halted_session
 * Inputs:
 *    session   The session whose state is being checked.
 * Outputs:
 *    <result>  zero if any process in the specified session is not
 *              secure halted, any other value if all processes in
 *              the specified session are secure halted.
 * Description:
 *    Returns whether a specified session is secure halted.
 ********************************************************************
 */
extern int is_secure_halted_session(int session);

/*
 ********************************************************************
 * Function:
 *    secure_halt_session
```

```
 * Inputs:
 *     session    The session which is being secure halted.
 * Outputs:
 *     <none>
 * Description:
 *     Places a session in the secure halt mode.  All tasks belonging
 *     to the specified session will be secure halted.  If any of the
 *     tasks are currently executing, this function will call the
 *     scheduler, which will swap out the tasks.
 ********************************************************************
 */
extern int secure_halt_session(int session);

/*
 ********************************************************************
 * Function:
 *     secure_unhalt_session
 * Inputs:
 *     session    The session which is being secure unhalted.
 * Outputs:
 *     <none>
 * Description:
 *     Removes a session from the secure halt mode.  All tasks
 *     belonging to the specified session will be secure unhalted.
 ********************************************************************
 */
extern int secure_unhalt_session(int session);

#endif /* defined(_SECURE_HALT_SESSION_H) */
```

2.   kernel/secure_halt_session.c

```
/*
 ********************************************************************
 * File: kernel/secure_halt_session.c
 *
 * Description:
 *     This is the implementation file for code which places sessions
 *     into and out of the secure halt mode.
 *
 * 2001-04-05  Created by Jerome Brock.
 ********************************************************************
 */

#include <linux/secure_halt_session.h>
#include <linux/secure_halt.h>
#include <linux/sched.h>

/*
 ********************************************************************
 * Function:
 *     is_secure_halted_session
 * Inputs:
```

50

```
 *      session    The session whose tasks are being checked.
 * Outputs:
 *      <result>   SEC_HALT_TRUE if every task in session is
 *                     secure halted.
 *                 SEC_HALT_FALSE if every task in session is not
 *                     secure halted.
 *                 SEC_HALT_ERROR_INVALID_SESSION if session is not a
 *                     valid session ID.
 * Description:
 *      Returns whether all tasks in a session are secure halted.
 ********************************************************************
 */
int is_secure_halted_session(int session){
        struct task_struct *p;
        int found_session = SEC_HALT_FALSE;

        for_each_task(p){
                if (p->session == session){
                        found_session = SEC_HALT_TRUE;

                        if (is_secure_halted(p) != SEC_HALT_TRUE){
                                return(SEC_HALT_FALSE);
                        }
                }
        }

        if (found_session){
                return(SEC_HALT_TRUE);
        }
        else{
                return(SEC_HALT_ERROR_INVALID_SESSION);
        }
}


/*
 ********************************************************************
 * Function:
 *      secure_halt_session
 * Inputs:
 *      session    The session whose tasks are being secure halted.
 * Outputs:
 *      <result>   SEC_HALT_SUCCESS if every task in session was
 *                     secure halted.
 *                 SEC_HALT_ERROR_INVALID_SESSION if session is not a
 *                     valid session ID.
 * Description:
 *      Places a session in the secure halt mode.  All tasks belonging
 *      to the specified session will be secure halted.  If any of the
 *      tasks are currently executing, this function will invoke the
 *      scheduler.
 ********************************************************************
 */
int secure_halt_session(int session){
```

```c
        // This does not call secure_halt_task for efficiency reasons, i.e
        // we only want one call to lock and unlock the task list

        int proc_is_executing = SEC_HALT_FALSE;
        struct task_struct *p;
        int found_session = SEC_HALT_FALSE;

        write_lock(&tasklist_lock);

        for_each_task(p){

                if (p->session == session){
                        p->secure_halted = SEC_HALT_TRUE;
                        found_session = SEC_HALT_TRUE;

                        if (p == current){
                                proc_is_executing = SEC_HALT_TRUE;
                        }
                }
        }

        write_unlock(&tasklist_lock);

        if (proc_is_executing == SEC_HALT_TRUE){
                schedule();
        }

        if (found_session == SEC_HALT_TRUE){
                return(SEC_HALT_SUCCESS);
        }
        else{
                return(SEC_HALT_ERROR_INVALID_SESSION);
        }
}

/*
 *******************************************************************
 * Function:
 *     secure_unhalt_session
 * Inputs:
 *     session    The session whose tasks are being secure unhalted.
 * Outputs:
 *     <result>   SEC_HALT_SUCCESS if every task in session was
 *                    secure unhalted.
 *                SEC_HALT_ERROR_INVALID_SESSION if session is not a
 *                    valid session ID.
 * Description:
 *     Removes a session in the secure halt mode.  All tasks belonging
 *     to the specified session will be secure unhalted.
 *******************************************************************
 */
int secure_unhalt_session(int session){
        // This does not call secure_unhalt_task for efficiency reasons,
i.e
```

52

```
        // we only want one call to lock and unlock the task list.

        struct task_struct *p;
        int found_session = SEC_HALT_FALSE;

        write_lock(&tasklist_lock);

        for_each_task(p){

                if (p->session == session){
                        found_session = SEC_HALT_TRUE;
                        p->secure_halted = SEC_HALT_FALSE;
                }
        }

        write_unlock(&tasklist_lock);

        if (found_session == SEC_HALT_TRUE){
                return(SEC_HALT_SUCCESS);
        }
        else{
                return(SEC_HALT_ERROR_INVALID_SESSION);
        }
}
```

D.   SECURE_HALT_ROUTING_MODULE

The SECURE_HALT_ROUTING_MODULE is implemented by adding code to

the existing Linux scheduler [TOR92a].

1.   kernel/sched.c

a.   Added

```
#include <linux/secure_halt.h>
```

b.   Changed the schedule() function

- *From:*

```
while (p != &init_task) {
      if (can_schedule(p))
      {
              int weight = goodness(prev, p, this_cpu);
              if (weight > c)
                      c = weight, next = p;
      }
      p = p->next_run;
}
```

- *To:*

```
while (p != &init_task) {
      if ((is_secure_halted(p) != SEC_HALT_TRUE)
            && (can_schedule(p)))
      {
            int weight = goodness(prev, p, this_cpu);
            if (weight > c)
                  c = weight, next = p;
      }
      p = p->next_run;
}
```

APPENDIX C. TESTING FRAMEWORK

A.   GOAL

The goal in developing a testing framework was to enable a user to selectively

halt and unhalt the session associated with a given virtual terminal. The framework

consisted of adding a new option to the "magic system request" facility and creating a

module that toggles the session on the current virtual terminal into and out of the secure

halt state. The magic system request facility is a specific combination of keystrokes that

the keyboard driver reroutes to the SYSTEM_REQUEST module for processing instead

of passing them through the usual path [MAR77]. The magic system request key

combination is "Alt-SysRq" and the specific system request option. The option added for

the testing framework was "Alt-SysRq-X."

B.   SOURCE CODE

1.   drivers/char/sysrq.c

a.   Added

```
#include <linux/secure_halt_test.h>
```

b.   Added to function handle_sysrq

```
switch (key) {

    case 'x':                              /* X -- Secure Halt Test */
        if (tty)
            toggle_tty_secure_halt(tty);
        break;

    . . .
}
```

2.   include/linux/secure_halt_test.h

```
/*
********************************************************************
```

55

```
 * File: include/linux/secure_halt_test.h
 *
 * Description:
 *     This is the header file for code which tests the operation of
 *     the secure halt mechanism.
 *
 * 2001-03-26  Created by Jerome Brock
 ********************************************************************
 */

#ifndef _SECURE_HALT_TEST_H
#define _SECURE_HALT_TEST_H

#include <linux/tty.h>

/*
 ********************************************************************
 * Function:
 *     toggle_tty_secure_halt
 * Inputs:
 *     tty         The tty whose processes are having their secure halt
 *                 state toggled.
 * Outputs:
 *     <none>
 * Description:
 *     Toggles the secure halt state of processes that are part of the
 *     specified tty's session.  If the session is currently secure
 *     halted, it will be unhalted, and vice versa.
 ********************************************************************
 */
void toggle_tty_secure_halt(struct tty_struct *tty);


#endif /* defined(_SECURE_HALT_TEST_H) */
```

3.   drivers/char/secure_halt_test.c

```
/*
 ********************************************************************
 * File: drivers/char/secure_halt_test.c
 *
 * Description:
 *     This is the implementation file for code which tests the
 *     operation of the secure halt mechanism.
 *
 * 2001-03-26  Created by Jerome Brock
 ********************************************************************
 */

#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/secure_halt.h>
#include <linux/secure_halt_session.h>
```

56

```c
#include <linux/secure_halt_test.h>

#define INTER_INVOCATION_SEC 2


/*
 *******************************************************************
 * Function:
 *     toggle_tty_secure_halt
 * Inputs:
 *     tty         The tty whose processes are having their secure halt
 *                 state toggled.
 * Outputs:
 *     <none>
 * Description:
 *     Toggles the secure halt state of processes that are part of the
 *     specified tty's session.  If the session is currently secure
 *     halted, it will be unhalted, and vice versa.
 *******************************************************************
 */
void toggle_tty_secure_halt(struct tty_struct *tty){

        static time_t last_invocation = 0;

        int tty_num = MINOR(tty->device);
        time_t curr_time = CURRENT_TIME;

        /* only allow invocation every so many seconds */

        if ((curr_time - last_invocation) < INTER_INVOCATION_SEC){
                return;
        }

        last_invocation = curr_time;

        printk("secure_halt_test:\n");

        if (is_secure_halted_session(tty->session)){
                secure_unhalt_session(tty->session);
                printk("    unhalted session %i on tty%i\n",
                        tty->session, tty_num);
        }
        else{
                secure_halt_session(tty->session);
                printk("    halted session %i on tty%i\n",
                        tty->session, tty_num);
        }

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[BEC98]   Beck, Michael, *et al.*, *Linux Kernel Internals*, Second Edition, Addison-Wesley, New York, 1998.

[CAR98]   Card, Rémy, Eric Dumas, Frank Mével, *The Linux Kernel Book*, West Sussux, John Wiley and Sons, 1998.

[GAS88]   Gasser, Morrie, *Building a Secure Computer System*, Van Nostrand Reingold, New York, 1988.

[GLE00]   Gleditsch, Arnie Georg, Per Kristian Gjermshus, *Cross-Referencing Linux*, http://lxr.linux.no, 2000.

[JOH00]   Johnson, Michael K., *et al.*, *procsps*, version 2.0.7, 2000.

[MAR77]   Mares, Martin, *Linux Kernel 2.2.18*, drivers/char/sysrq.c, 1977.

[MAX99]   Maxwell, Scott, *Linux Core Kernel Commentary*, Coriolis Group, Scottsdale, Arizona, 1999.

[MIC00]   Microsoft, *Windows 2000 Online Help*, Microsoft Corporation, 2000.

[PAR72]   Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM, Association for Computing Machinery*, December 1972.

[PLF97]   Pfleeger, Charles S., *Security in Computing*, Second Edition, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

[SCH75]   Schroeder, Michael D., "Engineering a Security Kernel for Multics," *Proceedings of the Fifth Symposium in Operating System Principles*, Association of Computing Machinery, 1975.

[SIL98]   Silberschatz, Abraham, Peter Baer Galvin, Operating System Concepts, Fifth Edition, Addison-Wesley, Berkeley, California, 1998.

[SOL98]   Soloman, David A., *Inside Windows NT*, Second Edition, Microsoft Press, Redmond, Washington, 1998.

[SUN99]   Sun Microsystems, Trusted Solaris User's Guide, version 7, 1999.

[TOR92a]   Torvalds, Linus, *Linux Kernel 2.2.18*, kernel/sched.c, 1992.

[TOR92b]   Torvalds, Linus, *Linux Kernel 2.2.18*, include/linux/sched.h, 1992.

[VMW01]   WMware, *WMware Workstation Product Specification*,
           http://www.vmware.com/pdf/detailed-specs-linux.pdf, 2001.

[WAN98]   Wang Government Services, *XTS-300 User's Manual, STOP 4.4.2*, 1998.

[WAR01]   Ward, Brian, *The Linux Kernel Howto*, v2.0, 2001.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
    8725 John J. Kingman Rd., STE 0944
    Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
    Naval Postgraduate School
    411 Dyer Rd.
    Monterey, California 93943-5101

3   Chairman, Code CS 1
    Computer Science Department
    Monterey, California 93943-5000

4. Dr. Cynthia E. Irvine 3
    Computer Science Department Code CS/Ic
    Monterey, California 93943-5000

5. Mr. Paul C. Clark 3
    Computer Science Department Code CS/Cp
    Monterey, California 93943-5000

6. Carl Siel 1
    Space and Naval Warfare Systems Command
    PMW 161
    Building OT-1, Room 1024
    4301 Pacific Highway
    San Diego, CA 92110-3127

7. Commander, Naval Security Group Command 1
    Naval Security Group Headquarters
    9800 Savage Road
    Suite 6585
    Fort Meade, MD 20755-6585

8. Ms. Deborah M. Cooper 1
    Deborah M. Cooper Company
    P.O. Box 17753
    Arlington, VA 22216